

# Towards A Generic Formal Framework for Access Control Systems

Jason Crampton<sup>1</sup> and Charles Morisset<sup>2\*</sup>

<sup>1</sup> Royal Holloway, University of London

Jason.Crampton@rhul.ac.uk

<sup>2</sup> Istituto di Informatica e Telematica (IIT), C.N.R.

Charles.Morisset@iit.cnr.it

**Abstract.** There have been many proposals for access control models and authorization policy languages, which form the basis for the design of access control systems. Most, if not all, of these proposals impose restrictions on the implementation of access control systems, limiting *inter alia* the type of authorization requests that can be processed or the structure of the authorization policies that can be specified. In this paper, we provide a formal characterization of the features of an access control system. However, we impose few restrictions on the actual implementation of such a system. In adopting this approach, we are able to define and articulate a number of desirable properties of an access control system in terms of our model, thereby laying the foundations for formal methods that can be used to analyze access control systems. We also illustrate that concrete instantiations of the framework yield important insights into the behavior of existing proposals for access control systems.

## 1 Introduction

One of the fundamental security requirements in modern computer systems is *access control*, a mechanism for constraining the interaction between (authenticated) users and protected resources. Generally, access control is enforced by a trusted component (historically known as the *reference monitor*), which is typically implemented by two functions: an *authorization enforcement function* (AEF) and an *authorization decision function* (ADF). The AEF traps all attempts by a user to interact with a resource (usually known as a *user request*) and transforms that request into one or more *authorization queries* (also known as *authorization requests*) which are forwarded to the ADF.

Most access control systems are policy-based. That is, an administrator specifies an authorization policy, which, in its simplest form, encodes those authorization requests that are authorized. The ADF takes an authorization query and an authorization policy as input and returns an authorization decision. For

---

\* Work partially supported by EU FP7-ICT project NESSoS (Network of Excellence on Engineering Secure Future Internet Software Services and Systems) under the grant agreement n. 256980.

this reason, it is common to refer to the AEF and ADF as the *policy enforcement point* (PEP) and *policy decision point* (PDP), respectively; it is this terminology that we will use henceforth.

An authorization policy is merely an encoding of the access control requirements of an application using the authorization language that is understood by the PDP. It is necessary, therefore, to make a clear distinction between an *abstract policy* and a *realizable policy*: the former is an arbitrary function from requests to decisions; the latter is a function that can be evaluated by the PDP. Given a particular policy language, there might be some abstract policies that are not realizable, which may be a limitation of the policy language in practice. The access control system used in early versions of Unix, for example, is rather limited. An important consideration, therefore, when designing an access control system is the *expressivity* of the policy language.

The increasing prevalence of open, distributed, computing environments, means that we may not be able to rely on a centralized authentication function to identify authorized users. This means that authorization decisions have to be made on the basis of user attributes (rather than user identities). In turn, this means that the format of authorization queries needs to be rather more flexible than that used in closed, centralized environments. The draft XACML 3.0 standard, for example, uses a much “looser” query format than its predecessor XACML 2.0. If we have no control over the attributes that are presented to the PDP, then a malicious user (or even a user who wishes to preserve the secrecy of some attributes) may be able to generate authorization decisions that are more “favorable” by withholding attributes from the PEP. A second important consideration, therefore, is whether authorization policies are guaranteed to be “monotonic” in the sense that providing fewer attributes in an authorization query yields a less favorable outcome.

*Related Work and Motivation.* There is an extensive literature on languages for specifying authorization policies. Much of this work can be traced back to the early work of Woo and Lam, which considered the possibility that different policies might evaluate to different authorization decisions [18]. More recent work has considered larger sets of policy decisions or more complex policy operators (or both) [2,4,5,6,7,8,9,12,13,14,15,16,17].

However, most of these proposals focus on closed systems in which requests have a fixed format, typically based on the subject-object-action paradigm. Moreover, few proposals attempt to look at the whole system, focusing instead on the details of the policy language. The two notable exceptions are XACML [16] and PTaCL [8]: the former provides a rich set of language features but lacks formality, while the latter provides formal semantics for policy evaluation, with some restrictions on what policies can express. However, both XACML and PTaCL make assumptions about the structure of policies: like many proposals in the literature, XACML and PTaCL are “target-based” [4,5,6,7,8,15,16,17].

The diversity of computing environments and the increasing automation of business processes make it necessary to support a wide variety of authorization policies. It seems unlikely, for example, that the access control requirements of

a social network will have much in common with those of an enterprise system. We believe, therefore, that it is important to design a framework that makes no assumptions about policy structure, policy decisions or policy operators. The framework should allow for the comparison of different design choices: What happens if I include decisions that can distinguish between missing information and non-matching information, or between missing information and malformed information? What happens if I use one particular method for combining the results obtained from evaluating sub-policies rather than some other method? And how do these methods interact with different choices for the set of decisions? The framework should also allow for the analysis of the policies that are expressible and to identify those that are not.

*Contributions.* In this paper, we develop an abstract framework for access control systems. This framework includes the structure of authorization queries, the specification of authorization policies and the evaluation of requests with respect to policies. Informally, we regard an authorization policy as a formula in some multi-valued propositional logic. A “policy formula” contains “policy variables” which are *atomic policies*; atomic in the sense that they are indivisible. Atomic policies are combined using *policy operators* defined over a set of *policy decisions* (which correspond to truth values taken from the set of values defined by the logic). For every request, a *policy decision* is assigned to each atomic policy. The valuation on atomic policies is extended to a valuation on policy formulae by recursively evaluating the sub-formulae and combining the values using the policy operators.

A number of these facets of our framework have appeared in the literature in some form: atomic policies are rather similar to the base policies defined by Crampton and Huth [6]; we have already noted that there are many proposals in the literature for policy operators defined over multi-valued decisions; and XACML and PTaCL provide frameworks for the specification and evaluation of a particular class of policies. However, the combination of these facets within a formal framework in order to reason about access control systems, the rigor of the semantics and the generality of the framework are, to the best of our knowledge, unique.

We also show that the framework is not merely of technical interest. In particular, we demonstrate that particular instantiations of our framework give rise to access control systems that are very similar to proposals in the literature such as XACML. Hence, we are able to examine various proposals in the literature and evaluate their suitability as building blocks for access control systems.

*Structure of the Paper.* In the next section we describe our framework for specifying access control systems. In Sec. 3 we investigate properties of an access control system, focusing on expressivity and monotonicity. In Sec. 4 we instantiate our framework and illustrate some of the problems that may arise in target-based policy languages and access control systems such as PTaCL [8] and XACML [15,16], two “target-based” policy languages. In the final section of the paper we summarize our contributions and describe our plans for future work.

## 2 The Framework

In this section we describe the various components of our framework before introducing the formal definition of an access control system (Def. 3). The components of the framework are illustrated in Appendix B using information flow and role-based access control policies.

### 2.1 Authorization Queries

An authorization query defines the representation by which user requests will be presented to the policy decision point (PDP). Typically, some intermediate component of the access control architecture will transform a user request into one or more authorization queries. In XACML – which provides a standardized reference architecture for access control [15, Figure 1] – this component is called the *context handler*, which will extract information from the user request and, optionally, seek information from other components of the access control architecture (*policy information points* in XACML) in order to construct an authorization query that can be processed by the PDP.

We assume throughout, as in XACML, that *the PDP only takes the authorization query and the policy as input*. In other words, the PDP communicates only with the context handler and requests no additional information from external sources.

We assume that the PDP recognizes some fixed set of data types  $\mathcal{T}$  and that each data type  $t \in \mathcal{T}$  can take some set of values, denoted by  $\text{dom}(t)$ . We also assume that for each type  $t \in \mathcal{T}$  and for each pair of values  $x$  and  $y$  of type  $t$ , the PDP can evaluate whether  $x$  equals  $y$ , and can recognize that a value  $x$  claimed to be of type  $t$  does not belong to  $\text{dom}(t)$ . Finally, we assume that the PDP can evaluate whether a value  $x$  of type  $t$  appears in a set of values, each of which is of type  $t$ .

There is considerable freedom in the way in which an authorization query might be represented. Typically, the representation will depend on where certain information relevant to access control is assumed to reside. It may be the case, for example, that one implementation might have the context handler push some information relevant to policy evaluation to the PDP, while in a different implementation that information might be encoded in the policy itself. We will assume that an authorization query is represented using attributes of the requesting user, the requested resource and the requested interaction between the user and the resource. An authorization query may also include environmental information such as the time at which the requested access was made or the location of the user. More formally, we introduce the following definition.

**Definition 1** *Let  $\mathcal{N}$  denote a set of attribute names. Let each attribute name  $\alpha$  be associated with a type  $\text{type}(\alpha)$  and let each type  $t$  be associated with a set of values  $\text{dom}(t)$  that can be taken by attributes of type  $t$ . Then an authorization request is a set  $\{(\alpha_1, v_1), \dots, (\alpha_m, v_m)\}$ , where  $v_i \in \text{dom}(\text{type}(\alpha_i))$  for all  $i$ .*

Henceforth, when no confusion can arise, we write  $\text{dom}(\alpha)$  rather than  $\text{dom}(\text{type}(\alpha))$ . We write  $\mathcal{Q}(\mathcal{N})$  to denote the set of possible requests, omitting  $\mathcal{N}$  when it is obvious from context. The definition above does not preclude the possibility that  $\alpha_i = \alpha_j$  for some  $i \neq j$ . That is, a query may contain two or more attributes with the same name. We show, in Sec. 4, that multiple attributes values may have undesirable implications for policy evaluation and discuss a more restricted form of requests.

## 2.2 Policy Decisions

We consider a set of decisions  $\text{Dec}$ , which corresponds to the set of constants in a multi-valued logic. Typically, we will include  $1_P$  (“allow”) and  $0_P$  (“deny”), which correspond to 1 (true) and 0 (false), respectively, in standard propositional logic.

It is common to consider a third value indicating that it is not possible (or not appropriate) to assign a conclusive decision to a policy. Typically, this is used when policies are applicable only to some strict subset of the set of all requests. Such a decision is particularly useful for attaching a meaning to policy sub-formulae, which is then resolved to a conclusive decision by the application of some policy operator. In XACML, this third decision is called “not applicable”, which we will denote by  $\perp_P$ .

Conversely,  $\top_P$  may be returned by a policy operator to indicate that its arguments conflict in some sense. We might define an operator  $\oplus$ , such that  $d_1 \oplus d_2 = \top_P$  if  $d_1, d_2 \in \{1_P, 0_P\}$  and  $d_1 \neq d_2$ . Such an operator could be used to indicate that two policies  $p_1$  and  $p_2$  returned different conclusive decisions.

We can also associate side effects with decisions. In this way, we can encode obligations that are attached with allowing or denying certain authorization queries. Then the semantics for policy operators must define how sets of obligations are combined in addition to defining how policy decisions are combined. Space limitations prevent us from exploring this matter further here.

## 2.3 Atomic Policies

An atomic policy is some representation of an authorization policy. The main requirement of an atomic policy is that the PDP can furnish the policy with a truth value (i.e. a decision) by inspection of the authorization query. Atomic policies may take many different forms, depending on the way in which the authorization query is represented and the desired authorization policy. In Appendix B, we illustrate two different forms of atomic policy that could be used to realize different types of authorization requirements.

Given a set of authorization queries  $\mathcal{Q}$ , a set of atomic policies  $\mathcal{A}$  and a set of decisions  $\text{Dec}$ , a *valuation function*  $\text{Eval} : \mathcal{Q} \times \mathcal{A} \rightarrow \text{Dec}$  is used to associate atomic policies with decisions. The valuation function would form part of the functionality of the PDP and it is here that the assumptions about the abilities of the PDP are crucial: for an atomic policy  $p$  and a request  $q$ ,  $\text{Eval}(q, p)$  will be defined by examining attribute values in the request.

It is important to note that even for very simple atomic policies, the definition of the valuation function is a rather delicate matter. In particular, the interpretation of missing attributes and of “junk” attribute values has a significant impact on policy evaluation. We return to this matter in Sec. 3 and Sec. 4.

## 2.4 Policy Operators

Given a set of atomic policies  $\mathcal{A}$ , we can build more complex policies from atomic policies using *policy operators*, each of which has the form  $\oplus : \text{Dec}^k \rightarrow \text{Dec}$  for some integer  $k \geq 1$ . Table 1 defines some particularly useful unary and binary operators. A policy operator could be included in a policy language for a variety of a different reasons.

- A policy operator may be used to combine, and possibly resolve conflicts between, the decisions returned by two (or more policies).  
We define binary operators  $\wedge_P$  and  $\vee_P$ , which act in the same way as logical conjunction and logical disjunction in Kleene’s 3-valued logic, respectively. We also define binary operators  $\Delta_P$  and  $\nabla_P$ , which have similar semantics to  $\wedge_P$  and  $\vee_P$  for conclusive decisions ( $1_P$  and  $0_P$ ), and essentially ignore the value  $\perp_P$ ; for example,  $\perp_P \wedge_P 1_P = \perp_P$ , while  $\perp_P \Delta_P 1_P = 1_P$ . These operators act in a way similar to the deny-overrides and allow-overrides algorithms in XACML.
- A policy operator may be used to restrict the scope of a policy and to determine the outcome of policy evaluation if a request is within scope. XACML targets may be interpreted in this way.  
We define a binary operator  $?_P$ , where  $p_1 ?_P p_2$  returns the evaluation of  $p_2$  if the evaluation of  $p_1$  returns  $1_P$  and returns  $\perp_P$  otherwise.
- A policy operator  $\neg_P$  may be used to reverse policy evaluation (swapping the decisions  $1_P$  and  $0_P$ ). We might include such an operator if we wish to provide a rich policy language that can be used to define arbitrarily complex policies.
- A policy operator may be used to transform a policy with “gaps” (one that is not defined for some requests) into one without gaps. Deny-by-default and allow-by-default are examples of such operators. For instance, the unary operator  $\sim_P$  has the effect of removing  $\perp_P$  by simply converting it to a deny: if policy  $p$  evaluates to  $\perp_P$ , then policy  $\sim_P p$  evaluates to  $0_P$ .

**Definition 2** *Given a set of atomic policies  $\mathcal{A}$ , a set of decisions  $\text{Dec}$  and a set of operators  $\text{Ops}$ , the set of realizable policies  $\mathcal{P}(\mathcal{A}, \text{Dec}, \text{Ops})$  is defined inductively:*

- $a \in \mathcal{P}(\mathcal{A}, \text{Dec}, \text{Ops})$  for every  $a \in \mathcal{A}$ ;
- for all  $p_1, \dots, p_k \in \mathcal{P}(\mathcal{A}, \text{Dec}, \text{Ops})$  and for every  $k$ -ary operator  $\oplus \in \text{Ops}$ ,  $k \geq 1$ ,  $\oplus(p_1, \dots, p_k) \in \mathcal{P}(\mathcal{A}, \text{Dec}, \text{Ops})$ .

**Table 1.** Operators over  $\{1_P, 0_P, \perp_P\}$

$d_1$	$d_2$	$\neg_P d_1$	$\sim_P d_1$	$d_1 \wedge_P d_2$	$d_1 \Delta_P d_2$	$d_1 \vee_P d_2$	$d_1 \nabla_P d_2$	$d_1 ?_P d_2$	$d_1 \bowtie_P d_2$
$1_P$	$1_P$	$0_P$	$1_P$	$1_P$	$1_P$	$1_P$	$1_P$	$1_P$	$1_P$
$1_P$	$0_P$	$0_P$	$1_P$	$0_P$	$0_P$	$1_P$	$1_P$	$0_P$	$\perp_P$
$1_P$	$\perp_P$	$0_P$	$1_P$	$\perp_P$	$1_P$	$1_P$	$1_P$	$\perp_P$	$\perp_P$
$0_P$	$1_P$	$1_P$	$0_P$	$0_P$	$0_P$	$1_P$	$1_P$	$\perp_P$	$\perp_P$
$0_P$	$0_P$	$1_P$	$0_P$	$0_P$	$0_P$	$0_P$	$0_P$	$\perp_P$	$0_P$
$0_P$	$\perp_P$	$1_P$	$0_P$	$0_P$	$0_P$	$\perp_P$	$0_P$	$\perp_P$	$\perp_P$
$\perp_P$	$1_P$	$\perp_P$	$0_P$	$\perp_P$	$1_P$	$1_P$	$1_P$	$\perp_P$	$\perp_P$
$\perp_P$	$0_P$	$\perp_P$	$0_P$	$0_P$	$0_P$	$\perp_P$	$0_P$	$\perp_P$	$\perp_P$
$\perp_P$	$\perp_P$	$\perp_P$	$0_P$	$\perp_P$	$\perp_P$	$\perp_P$	$\perp_P$	$\perp_P$	$\perp_P$

Given  $\mathcal{A}$ ,  $\text{Dec}$  and a valuation function  $\text{Eval} : \mathcal{A} \rightarrow \text{Dec}$ , we define the extended valuation function  $\text{Eval}^+ : \mathcal{Q} \times \mathcal{P}(\mathcal{A}, \text{Dec}, \text{Ops}) \rightarrow \text{Dec}$  for the set of realizable policies as follows:

$$\text{Eval}^+(q, p) = \begin{cases} \text{Eval}(q, p) & \text{if } p \in \mathcal{A}, \\ \oplus(\text{Eval}^+(q, p_1), \dots, \text{Eval}^+(q, p_k)) & \text{if } p = \oplus(p_1, \dots, p_k). \end{cases}$$

## 2.5 Access Control System

**Definition 3** An access control system is a tuple  $(\mathcal{Q}, \mathcal{A}, \text{Dec}, \text{Ops}, \text{Eval})$ , where  $\mathcal{Q}$  is a set of authorization queries;  $\mathcal{A}$  is a set of atomic policies;  $\text{Dec}$  is a set of policy decisions;  $\text{Ops}$  is a set of policy operators; and  $\text{Eval} : \mathcal{Q} \times \mathcal{A} \rightarrow \text{Dec}$  is a valuation function for atomic policies.

Given a set of requests  $\mathcal{Q}$  and a set of decisions  $\text{Dec}$ , an arbitrary authorization policy may be regarded as a function  $\pi : \mathcal{Q} \rightarrow \text{Dec}$ . We call any such function an *abstract policy*.

**Definition 4** Given a system  $\mathcal{S} = (\mathcal{Q}, \mathcal{A}, \text{Dec}, \text{Ops}, \text{Eval})$  and an abstract policy  $\pi : \mathcal{Q} \rightarrow \text{Dec}$ , we say that  $\mathcal{S}$  realizes  $\pi$  if, and only if, there exists a policy  $p \in \mathcal{P}(\mathcal{A}, \text{Dec}, \text{Ops})$  such that for all requests  $q$ ,  $\pi(q) = \text{Eval}^+(q, p)$ .

When no confusion can arise, we may use the term policy to refer to either an abstract policy or a concrete policy written using  $\mathcal{A}$  and  $\text{Ops}$ . And when  $\mathcal{A}$ ,  $\text{Dec}$  and  $\text{Ops}$  are obvious from context, we will simply write  $\mathcal{P}$  for the set of realizable policies  $\mathcal{P}(\mathcal{A}, \text{Dec}, \text{Ops})$ .

## 3 General Properties

An access control system  $(\mathcal{Q}, \mathcal{A}, \text{Dec}, \text{Ops}, \text{Eval})$  may realize a large number of policies. It might therefore be useful to be able to characterize some general

properties of those policies. In particular, we want to characterize the notion of *monotonicity* for a policy, which indicates whether a request is more likely to be granted if more information is supplied. In this section, we also define the notion of *completeness* of a system, which is concerned with the expressivity of the policy operators. Finally, we outline some other properties, such as safety and static decision-based analysis.

### 3.1 Monotonicity

Informally, a policy is monotonic whenever removing information from a request does not lead to a “better” policy decision. Such a property is of particular relevance in open systems, where users might be able to control what information they supply to the access control system. A system in which all realizable policies are monotonic means that a malicious user gains no advantage by suppressing information when making a request.

We model information hiding using a partial ordering  $\leq_Q$  on  $Q$ ; the intuitive interpretation of  $q \leq_Q q'$  is that  $q$  contains less information than  $q'$ . We also need to specify what it means for a decision to “benefit” a user, and thus we assume the existence of an ordering relation  $\leq_P$  on  $\text{Dec}$ ; again, the intuitive interpretation of  $d_1 \leq_P d_2$  is that the decision  $d_2$  is of greater benefit than  $d_1$ .<sup>3</sup> In Sec. 4 we provide specific definitions for each of these relations.

**Definition 5** *Given a set of authorization queries  $(Q, \leq_Q)$  and a set of decisions  $(\text{Dec}, \leq_P)$ , an abstract policy  $\pi : Q \rightarrow \text{Dec}$  is monotonic if, and only if for all  $q, q' \in Q$ :*

$$q \leq_Q q' \Rightarrow \pi(q) \leq_P \pi(q').$$

*We say that an access control system  $(Q, \mathcal{A}, \text{Dec}, \text{Ops}, \text{Eval})$  is monotonic if every realizable policy is monotonic.*

**Definition 6** *An operator  $\oplus : \text{Dec}^k \rightarrow \text{Dec}$  is monotonic if and only if, for all  $d_1, \dots, d_k \in \text{Dec}$  and  $d'_1, \dots, d'_k \in \text{Dec}$  with  $d'_i \leq_P d_i$ ,  $1 \leq i \leq k$ ,*

$$\oplus(d'_1, \dots, d'_k) \leq_P \oplus(d_1, \dots, d_k).$$

Informally, a binary operator  $\oplus$  is monotonic if increasing the truth value in any one of its arguments has the effect of increasing the truth value of the computation.

**Proposition 7** *An access control system  $(Q, \mathcal{A}, \text{Dec}, \text{Ops}, \text{Eval})$  is monotonic if, and only if every policy in  $\mathcal{A}$  is monotonic and every operator in  $\text{Ops}$  is monotonic.*

*Proof.* The result follows by a standard induction over the structure of policies.

In Sec. 4, we instantiate our framework with a particular definition of atomic policies. By assuming some restrictions on the structure of requests, we can prove the monotonicity of a system using results from partial logic [3].

---

<sup>3</sup> Note that we consider this relation to be statically defined over decisions, and to be independent of the request.



### 3.2 Completeness

As we noted above, any realizable policy  $p \in \mathcal{P}$  corresponds to an abstract policy  $\pi : \mathcal{Q} \rightarrow \text{Dec}$ . However, in general, the converse is not necessarily true: some abstract policies might not be realizable by a given system. Trivially, for example, a system without any atomic policies does not realize any abstract policy. It follows that the set of abstract policies that can be realized by a system represents an intuitive notion of expressivity. A system that can realize every every abstract policy is said to be complete. More formally:

**Definition 8** *An access control system  $(\mathcal{Q}, \mathcal{A}, \text{Dec}, \text{Ops}, \text{Eval})$  is complete if, and only if for any abstract policy  $\pi : \mathcal{Q} \rightarrow \text{Dec}$ , there exists a realizable policy  $p \in \mathcal{P}(\mathcal{A}, \text{Dec}, \text{Ops})$  such that for all  $q \in \mathcal{Q}$ ,  $\text{Eval}^+(q, p) = \pi(q)$ .*

The completeness of a system  $(\mathcal{Q}, \mathcal{A}, \text{Dec}, \text{Ops}, \text{Eval})$  will depend on the definition of atomic policies, the set  $\text{Ops}$  and  $\text{Eval}$ . Intuitively, in order to prove the completeness of a system, we need to have for each potential attribute  $\alpha$  and each value  $v$  an atomic policy that can distinguish  $(\alpha, v)$ . We propose in Sec. 4 a representation of atomic policies that can distinguish attribute name-value pairs in this way, from which we can prove a completeness result.

It is worth observing that in general, if a system is both monotonic and complete, then the ordering over requests is limited to the identity relation.

**Proposition 9** *Let  $\mathcal{S} = ((\mathcal{Q}, \leq_{\mathcal{Q}}), \mathcal{A}, (\text{Dec}, \leq_{\text{P}}), \text{Ops}, \text{Eval})$ , where  $|\text{Dec}| > 1$ , be a monotonic, complete system. Then  $\leq_{\mathcal{Q}}$  is the identity relation.*

*Proof.* Let us consider two requests  $q$  and  $q'$  such that  $q \leq_{\mathcal{Q}} q'$  and  $q \neq q'$ . Since  $|\text{Dec}| > 1$ , there exist  $d_1$  and  $d_2$  such that  $d_1 \not\leq_{\text{P}} d_2$ . Since  $\mathcal{S}$  is complete, there exists a policy  $p$  generated by  $\mathcal{S}$  such that  $\text{Eval}(p, q) = d_1$  and  $\text{Eval}(p, q') = d_2$ , which contradicts the monotonicity of  $p$ .

Informally, this result states that if we wish to have a (non-trivial) monotonic system then we cannot expect to have a complete system. Instead, what we should aim for is a system that realizes all *monotonic, abstract* policies, and such a system is said to be *monotonically-complete*.

### 3.3 Other properties

The notions of monotonicity and completeness are examples of general properties we can characterize. We believe that our framework is expressive enough to characterize many other useful properties. Due to space restrictions we only sketch here some properties of interest, leaving their full characterization and analysis for future work.

**Safety** An important question for an access control system is to know whether a particular access request is authorized at some point in the future, potentially

through several modifications of the security configuration. This is known as the *safety* problem, and has been shown to be undecidable in general [?].

In order to model the notion of safety in our framework, we first need to model the ability to change the security configuration. Hence, we introduce a new attribute `modif`, such that  $\text{dom}(\text{modif}) = \mathcal{P} \rightarrow \mathcal{P}$ , and we write  $\mu$  for values of `modif`. Intuitively, given a policy  $p$ ,  $\mu(p)$  stands for the policy obtained by applying the modification  $\mu$ .

Since a policy modification is simply an attribute value, then deciding whether such a modification is allowed or not can be directly managed by the policy itself. In other words, given a policy  $p$ , a request  $q$  and a modification  $\mu$  such that  $\mu$  belongs to  $q$ ,  $\text{Eval}^+(q, p) = 1_P$  means that the policy  $p$  can be modified to the policy  $\mu(p)$ . The safety problem is then equivalent to knowing, given a policy  $p$  and a request  $q$ , whether there exists a sequence of modifications over  $p$  such that  $q$  is allowed. We represent this concept with the relation  $\leadsto$ , where  $p \leadsto q$  informally means that  $p$  eventually enables  $q$ . Intuitively,  $p \leadsto q$  if, and only if, either  $p$  authorizes  $q$ , or there exists a modification  $\mu$  allowed by  $p$  (not necessarily with the request  $q$ ) such that  $\mu(p) \leadsto q$ . More formally, we have:

$$p \leadsto q \Leftrightarrow \left( \begin{array}{l} (\text{Eval}^+(q, p) = 1_P) \\ \vee \exists q' \in \mathcal{Q} \exists \mu \in \mathcal{P} \rightarrow \mathcal{P} (\mu \in q') \wedge (\text{Eval}^+(q', p) = 1_P) \wedge (\mu(p) \leadsto q) \end{array} \right)$$

where  $\vee$  and  $\wedge$  denote logical disjunction and conjunction, respectively. In general, the safety problem considers whether  $p \leadsto q$  holds for policy  $p$  and request  $q$ . Note that this relation  $\leadsto$  is not structurally inductive, since  $\mu(p)$  is not structurally smaller than  $p$ , and therefore  $\leadsto$  is in general undecidable (there might exist an infinite chain of modifications). However, in some cases, for instance when the number of basic permissions is finite, or when no modification of the policy is allowed, then the relation  $\leadsto$  may be computable.

**Static Decision-based Analysis** In addition to monotonicity and safety, there is a wide range of properties one might want to verify over a given policy by considering as a propositional formula. For instance, the satisfiability problem consists in checking whether there exists a valuation on the atomic policies such that the policy is true, in other words whether the policy allows at least one request. Such a problem can be extended to constraint satisfaction problems, for instance in order to know which attribute values are missing in a request in order to allow the request.

## 4 Attribute-Based Access Control

In this section we look at the framework in practice. In particular, we illustrate how the framework can be instantiated to produce a concrete access control system that implements attribute-based access control using target-based policies. Given a particular format for atomic policies, we explore how different restrictions on the request format and different choices for the `Eval` function give rise

to systems with greater expressive power, on the one hand, to systems with stronger guarantees about the evaluation of policies on the other.

Throughout this section we use a particularly simple, perhaps the simplest, representation of atomic policies. Given a set of attribute names  $\mathcal{N}$  and a domain  $\text{dom}(\alpha)$  for each attribute  $\alpha \in \mathcal{N}$ , we write  $\mathcal{A}(\mathcal{N})$  to denote the set of atomic policies  $\{(\alpha, v) : \alpha \in \mathcal{N}, v \in \text{dom}(\alpha)\}$ .

#### 4.1 A Monotonic System

In this subsection, we choose  $\text{Three} = \{1_P, 0_P, \perp_P\}$  for the set of decisions and define the ordering  $x \leq_3 y$  if and only if  $x = y$  or  $x = \perp_P$ . Note that other ordering relations are possible: we choose this particular one because it corresponds to the “degree-of-definedness” relation used in partial logic to reason about missing (truth) values [3]; this ordering corresponds in an obvious way with our wish to characterize missing information. We show that if we restrict the format of requests in a particular way then it is possible to construct a monotonic access control system.

**Requests.** A request  $q$  is a set of attribute name-value pairs  $\{(\alpha_1, v_1), \dots, (\alpha_m, v_m)\}$ , with the restriction that each attribute name may occur at most once in a request.<sup>4</sup> We write  $\mathcal{Q}_1$  for this set of requests. A request may be *incomplete*, in the sense that there may exist  $\alpha \in \mathcal{N}$  such that  $(\alpha_i, v_i) \neq \alpha$  for any  $i$ . And a request may be *malformed*, in the sense that there may exist  $(\alpha_i, v_i) \in q$  such that  $v_i \notin \text{dom}(\alpha_i)$ . In other words, a request may be missing certain attribute names and may include attribute values that do not belong to the domain of the attribute. We define the following order  $\leq_{\mathcal{Q}_1}$  over requests.

**Definition 10** *Given two requests  $q$  and  $q'$ ,  $q \leq_{\mathcal{Q}_1} q'$  if and only if, for all  $(\alpha, v) \in q$  such that  $v \in \text{dom}(\alpha)$ ,  $(\alpha, v) \in q'$ .*

It is consistent for us to define, for all requests  $\{(\alpha, v)\}$ ,  $\emptyset \leq_{\mathcal{Q}_1} \{(\alpha, v)\}$ , with equality if and only if  $v = \circ$ , since we may interpret the request  $\emptyset$  as  $\{(\alpha, \circ)\}$ . Note that  $\leq_{\mathcal{Q}_1}$  does not define a partial ordering on  $\mathcal{Q}_1$ , since it is not anti-symmetric.<sup>5</sup> However, it is easy to establish that the relation  $\equiv_{\mathcal{Q}_1}$ , where  $q \equiv_{\mathcal{Q}_1} q'$  if and only if  $q \leq_{\mathcal{Q}_1} q'$  and  $q' \leq_{\mathcal{Q}_1} q$ , is an equivalence relation.

**The  $\text{Eval}_1$  Function.** Recall that  $\text{Eval}_1$  assigns a value to atomic policies. We define  $\text{Eval}_1$  recursively over the length of the request. We interpret the empty request as  $(\alpha, \circ)$ , where  $\circ$  signifies both a missing attribute (a “hole”) and a

<sup>4</sup> That is  $\alpha_i = \alpha_j$  if and only if  $i = j$ .

<sup>5</sup> For example, if  $q = \{(\alpha, v)\}$  and  $q' = \{(\alpha, v), (\alpha', \circ)\}$ , then  $q \leq_{\mathcal{Q}_1} q'$  and  $q' \leq_{\mathcal{Q}_1} q$ , but  $q \neq q'$ .

value that does not belong to the domain of  $\alpha$ . We define

$$\text{Eval}_1(q, (\alpha, v)) = \begin{cases} 1_P & \text{if } q \ni (\alpha, v), \\ 0_P & \text{if } q \ni (\alpha, v') \text{ and } v' \in \text{dom}(\alpha) \text{ and } v' \neq v, \\ \perp_P & \text{otherwise.} \end{cases}$$

In other words, if the request contains a matching attribute name-value pair then  $1_P$  will be returned; if the request contains a matching attribute name and a non-matching (legitimate) attribute value then  $0_P$  will be returned; otherwise  $\perp_P$  will be returned. The policy  $(\text{actionID}, \text{read})$ , for example, will return  $1_P$  whenever the request contains the pair  $(\text{actionID}, \text{read})$ ; this corresponds to the **allRead** policy in Appendix B.

It is important to emphasize that we do not assume that the equality predicate necessarily returns either true or false. In particular, we consider the case that a request contains the pair  $(\alpha, \circ)$ , where  $\circ$  stands for any value that does not belong to  $\text{dom}(\alpha)$ . In this case, the truth value of  $v = \circ$  is indeterminate, in which case  $\text{Eval}_1(q, (\alpha, v)) = \perp_P$ .

Moreover, we have the following result (the proof of this proposition and other results, can be found in Appendix A).

**Proposition 11** *Let  $q, q' \in \mathcal{Q}_1$  such that  $q \equiv_{\mathcal{Q}_1} q'$ . Then for all atomic policies  $(\alpha, v)$ ,*

$$\text{Eval}_1(q, (\alpha, v)) = \text{Eval}_1(q', (\alpha, v)).$$

The above result tells us that all elements of the same equivalence class induce the same valuation on the set of atomic policies. Hence,  $\text{Eval}_1(q, (\alpha, v))$  the evaluation of a request is determined solely by the attributes to which legitimate values have been assigned. In the following, we work modulo the equivalence relation  $\equiv_{\mathcal{Q}_1}$ , with  $\leq_{\mathcal{Q}_1}$  being a partial order defined on the set of equivalence classes.

**Proposition 12** *For all requests  $q, q' \in \mathcal{Q}_1$  such that  $q \leq_{\mathcal{Q}_1} q'$  and for all atomic policies  $(\alpha, v) \in \mathcal{A}(\mathcal{N})$ ,*

$$\text{Eval}_1(q, (\alpha, v)) \leq_3 \text{Eval}_1(q', (\alpha, v)).$$

**Building Policies.** Using the policy operators  $\sim_P$ ,  $\neg_P$  and  $\vee_P$ , we can construct the following policies:  $\sim_P(\alpha, v)$  and  $(\alpha, v) \vee_P \neg_P(\alpha, v)$ ;  $\sim_P(\alpha, v)$  returns  $1_P$  if the request contains a matching attribute value  $v$  for attribute  $\alpha$  (and  $0_P$  otherwise), while  $(\alpha, v) \vee_P \neg_P(\alpha, v)$  returns  $1_P$  if the request contains the attribute  $\alpha$  and that a valid value is associated with it (and  $\perp_P$  otherwise). The latter policy can be used to identify well-formed requests. We are not aware of any other policy language or policy algebra, with the exception of PTaCL [8], that supports this type of “request interface” policy.

Similarly, the policy operators  $\Delta_P$  and  $\wedge_P$  can be used to construct useful policies such as  $(\alpha, v) \wedge_P (\alpha', v')$ , which evaluates to  $1_P$  only if both attributes

are present in the request and have the appropriate values, and  $(\alpha, v) \Delta_P (\alpha', v')$ , which evaluates to  $1_P$  if exactly one attribute is present with the correct value.<sup>6</sup>

**Proposition 13** *The system  $(\mathcal{Q}_1, \mathcal{A}(\mathcal{N}), \text{Three}, \{\neg_P, \wedge_P, \vee_P, \bowtie_P, ?_P\}, \text{Eval}_1)$  is monotonic.*

*Proof.* The result follows directly from Proposition 7, Proposition 12 and the fact that the operators  $\neg_P, \wedge_P, \vee_P, \bowtie_P$  and  $?_P$  are monotonic.

Note that the operators  $\sim_P, \Delta_P$  and  $\nabla_P$  are not monotonic, and therefore cannot be used to build a monotonic system.

**Proposition 14** *The system  $(\mathcal{Q}_1, \mathcal{A}(\mathcal{N}), \text{Three}, \{\neg_P, \wedge_P, \vee_P, \bowtie_P, ?_P\}, \text{Eval}_1)$  is monotonically complete.*

*Proof.* Let  $\pi$  be an abstract monotonic policy. We show that there exists a policy  $p(\pi) \in \mathcal{P}_1$  that realizes  $\pi$ .

$(\mathcal{Q}_1, \leq_{\mathcal{Q}_1})$  is a finite partially ordered set, so we may enumerate its elements using a topological sort. That is:  $\mathcal{Q}_1 = \{q_0, q_1, \dots, q_n\}$ , for some  $n$  determined by  $\mathcal{N}$  and  $\text{dom}(\alpha)$ ,  $\alpha \in \mathcal{N}$ ; and  $q_i \leq_{\mathcal{Q}_1} q_j$  implies that  $i \leq j$ . Since the empty request is less than any other request, we necessarily have  $q_0 = \emptyset$ .

For any non-empty request  $q = \{(\alpha_1, v_1), \dots, (\alpha_m, v_m)\}$ , we define the policy  $t_q = (\alpha_1, v_1) \wedge_P \dots \wedge_P (\alpha_m, v_m)$ . Note that  $\text{Eval}_1^+(q', t_q) = 1_P$  for all  $q' \geq q$ .

Now, for any request  $q \in \mathcal{Q}_1$ : either  $q = q_0$ , in which case  $\text{Eval}_1^+(q, t_{q_i}) = \perp_P$ , for all  $t_{q_i}$ , or  $q = q_i$ , for some  $i \geq 1$ ;  $\text{Eval}_1^+(q, t_{q_i}) = 1_P$ ; and, for all  $j > i$ ,  $\text{Eval}_1^+(q, t_{q_j}) \neq 1_P$ . In other words, this index  $i$  “identifies” the request  $q$ . We want to define an operator  $\oplus_\pi : \text{Three}^n \rightarrow \text{Three}$ , such that, given the evaluation of  $t_{q_1}, \dots, t_{q_n}$ , this operator “identifies” the request  $q_i$  corresponding to this evaluation (i.e. the maximal element equal to  $1_P$ ), and returns  $\pi(q_i)$ .

More formally, the operator  $\oplus_\pi$  takes a  $n$ -tuple of decisions  $(d_1, \dots, d_n)$ , such that each  $d_i = \text{Eval}_1^+(q, t_{q_i})$ , for some  $q$ , and is defined as:

$$\oplus_\pi(d_1, \dots, d_n) = \begin{cases} \pi(q_m) & \text{if } m = \max \{1 \leq i \leq n \mid d_i = 1_P\}, \\ \pi(\emptyset) & \text{otherwise.} \end{cases}$$

We now prove that  $\oplus_\pi$  is monotonic. Intuitively, we want to prove that the order over tuples of decisions implies the order over requests, in order to use the monotonicity of  $\pi$ . Let  $d_1, \dots, d_n \in \text{Three}$  and  $d'_1, \dots, d'_n \in \text{Three}$  with  $d'_i \leq_P d_i$ ,  $1 \leq i \leq n$ . Let  $q'$  and  $q$  be the requests identified by  $(d'_1, \dots, d'_n)$

<sup>6</sup> We can also define the policy  $(\alpha, v) \nabla_P (\alpha', v')$ , which evaluates to  $1_P$  if at least one of the attributes is present and has the required value. Note that we can, trivially, encode an access matrix using atomic policies: the matrix entry  $(s, o, a)$  may be encoded as  $\sim_P(\text{subjectID}, s) \wedge_P \sim_P(\text{objectID}, o) \wedge_P \sim_P(\text{actionID}, a)$ ; the matrix is encoded as the disjunction of all such authorized triples. In Sec. 4.2, we demonstrate that there is a strong correspondence between atomic policies and the targets used to define policies in XACML and PTaCL.

and  $(d_1, \dots, d_n)$ , respectively. By definition, we have  $\oplus_\pi(d'_1, \dots, d'_n) = \pi(q')$  and  $\oplus_\pi(d_1, \dots, d_n) = \pi(q)$ . Furthermore, let  $m$  be the maximal index such that  $d'_m = 1_P$ , it follows that  $q' = q_m$ . Since  $d'_m \leq_P d_m$ , we can deduce that  $d_m = 1_P$ , implying that  $q_m \leq_{Q_1} q$ , that is,  $q' \leq_{Q_1} q$ . By hypothesis,  $\pi$  is monotonic, and thus we have  $\pi(q') \leq_P \pi(q)$ , allowing us to conclude that  $\oplus_\pi$  is monotonic.

Finally, we know, by a result of Blamey [3], that any monotonic operator can be built from  $\{\neg_P, \wedge_P, \vee_P, \mathbb{X}_P\}$ . In other words, the policy  $\oplus_\pi(t_{q_1}, \dots, t_{q_n})$  belongs to  $\mathcal{P}_1$ , and therefore we can conclude that  $\pi$  can be realized by  $(Q_1, \mathcal{A}(\mathcal{N}), \text{Three}, \{\neg_P, \wedge_P, \vee_P, \mathbb{X}_P, ?_P\}, \text{Eval}_1)$ .

**Proposition 15** *The system  $(Q_1, \mathcal{A}(\mathcal{N}), \text{Three}, \{\neg_P, \sim_P, \vee_P\}, \text{Eval}_1)$  is complete.*

*Proof.* The structure of this proof is similar to that of Proposition 14, with the difference that we consider now the abstract policy  $\pi$  is not necessarily monotonic. We show that there exists a policy  $p(\pi) \in \mathcal{P}_1$  that realizes  $\pi$ .

Concretely, we consider the enumeration over the requests, and we build the same operator  $\oplus_\pi$ . However, in this case, we do not need to prove the monotonicity of  $\oplus_\pi$ , and we use instead the fact that the logic  $\{1_P, 0_P, \perp_P, \neg_P, \sim_P, \vee_P\}$  is functionally complete [8], which ensures that  $\oplus_\pi$  can be built from  $\{\neg_P, \sim_P, \vee_P\}$ .

The above propositions provide guidance to the system designer. She can choose, for example, between a system that realizes only and all monotonic policies, and a system in which all policies are realizable, but some may be non-monotonic. Clearly, the choice depends on the demands of the application and the constraints of the underlying environment. While we cannot make this choice for the policy designer, our framework can only help her make that decision.

## 4.2 XACML as an Instance of the Framework

XACML is a well known, standardized framework for access control systems and is best known for the language it defines for specifying policies. A policy written in XACML may be viewed as a tree in which (i) the leaves are XACML rules (ii) interior nodes whose children are all leaves are XACML policies, and (iii) all other nodes are XACML policy sets. Each XACML rule is associated with an *effect*, which is taken from the set  $\{1_P, 0_P\}$ . All XACML rules, policies and policy sets define a *target*, which is used to limit their respective applicability. The target language is expressed over the attributes of a request and involves the comparison of attribute values in the target with attribute values in the request. The effects of applicable child rules are combined to produce a decision for each policy; the decisions for each applicable policy and policy set are combined to produce a decision for each policy set.

Using  $\mathcal{A}(\mathcal{N})$  as the set of atomic policies and  $\text{Eval}_1$  as defined in Section 4.1, we note a strong correspondence between our framework and XACML. A policy of the form  $((\alpha_1, v_1) \wedge_P \dots \wedge_P (\alpha_m, v_m)) ?_P d$  encodes an XACML rule whose target matches those requests containing attribute values  $v_1, \dots, v_m$  for attributes

$\alpha_1, \dots, \alpha_m$ , respectively, and whose effect is  $d$ . Moreover, the operators  $\Delta_P$  and  $\nabla_P$  act, respectively, like the **deny-overrides** and **allow-overrides** algorithms used in XACML. The completeness of our framework allows us to assert that any of the policy-combining algorithms specified in XACML can be expressed in our framework.

The fact that XACML uses non-monotonic operators means that it is possible, in principle, to change the decision reached by a policy by withholding attributes from a request. Note, in particular, that  $1_P \Delta_P 0_P = 0_P$ , while  $1_P \Delta_P \perp_P = 1_P$ . This suggests that care should be exercised when authoring XACML policies because it may be possible for a malicious user, by withholding attribute information, to cause policy evaluation to return  $1_P$  rather than  $0_P$ .

### 4.3 A Non-Monotonic System

In practice, we may wish to include multiple values for the same attribute. The most obvious example arises in role-based access control systems, where a request may contain several role values. We now discuss the consequences of modifying the structure of requests in this way, and we write  $\mathcal{Q}_*$  for the set of such requests.

The previous function  $\text{Eval}_1$  is changed to the function  $\text{Eval}_*$ , such that  $\text{Eval}_*(q, (\alpha, v))$  returns  $1_P$  whenever  $(\alpha, v)$  belongs to  $q$ ,  $\perp_P$  if  $q$  contains no pair of the form  $(\alpha, v')$  for any  $v' \in \text{dom}(\alpha)$ , and  $0_P$  otherwise. Most attribute-based access control systems, including XACML 3.0 and PTaCL, choose to evaluate requests in this way. However, it is trivial to see that such a strategy is not monotonic with respect to the ordering  $\leq_{\mathcal{Q}_1}$  defined in Section 4.1. In particular, for  $v, v' \in \text{dom}(\alpha)$  with  $v \neq v'$ , we have  $\{(\alpha, v')\} \leq_{\mathcal{Q}_1} \{(\alpha, v), (\alpha, v')\}$ , but  $0_P = \text{Eval}_*(\{(\alpha, v')\}, (\alpha, v)) \not\leq_P \text{Eval}_*(\{(\alpha, v), (\alpha, v')\}, (\alpha, v)) = 1_P$ .

$\text{Eval}_*$  is monotonic for atomic policies if we adopt the ordering  $\perp_P < 0_P < 1_P$ . This means that omitting attributes can cause the evaluation of an atomic policy to change from  $1_P$  to  $0_P$  or  $\perp_P$ , or from  $0_P$  to  $\perp_P$ . While this seems to be reasonable, when combined with operators such as  $\Delta_P$ , omitting attributes can cause the evaluation of a policy to change from  $0_P$  to  $1_P$ . We established in our work on PTaCL that certain operators (some of which are not used in this paper) are monotonic, while others, such as  $\neg_P$  and  $\vee_P$  are not monotonic [8].

### 4.4 A System with a Four-Valued Decision Set

PTaCL is a recent proposal [8] that seeks to provide a stronger theoretical foundation for target-based policy languages such as XACML. PTaCL policies have a similar structure to those in XACML, although the operators used are somewhat different, in order to combat the problem of non-monotonicity that might arise in target evaluation. However, PTaCL runs into a different type of non-monotonicity problem because atomic policies are evaluated in the way described in the previous subsection.

The choice in the previous subsection to evaluate an atomic policy  $(\alpha, v)$  to  $1_P$  if  $(\alpha, v)$  and  $(\alpha, v')$  with  $v \neq v'$  and  $v, v' \in \text{dom}(\alpha)$  could be regarded as

being logically inconsistent in the sense that the request also contains a non-matching value. One could equally well argue, for example, that the request should evaluate to  $0_P$ .

In order to cope with such situations, we might, therefore, choose to work with the 4-valued logic  $\mathbf{Four} = \{1_P, 0_P, \perp_P, \top_P\}$ , using  $\top_P$  to denote conflicting information (in contrast to  $\perp_P$  which signifies lack of information). We introduce the ordering  $\leq_4$ , where  $d_1 \leq_4 d_2$  if, and only if,  $d_1 = \perp$ ,  $d_1 = d_2$  or  $d_2 = \top_P$ , and we define

$$\text{Eval}_4(q, (\alpha, v)) = \begin{cases} \top_P & \text{if } q \ni (\alpha, v), (\alpha, v') \text{ such that } v, v' \in \text{dom}(\alpha) \text{ and } v' \neq v, \\ 1_P & \text{if } q \ni (\alpha, v) \text{ and } q \not\ni (\alpha, v') \text{ such that } v' \neq v, \\ 0_P & \text{if } q \not\ni (\alpha, v) \text{ and } q \ni (\alpha, v') \text{ such that } v' \neq v, \\ \perp_P & \text{otherwise.} \end{cases}$$

The definition of the policy operators  $\nabla_P$  and  $\Delta_P$  can be extended to unary operators on  $\mathbf{Four}$ , where

$$\nabla_P d = \begin{cases} 1_P & \text{if } d = \top_P, \\ d & \text{otherwise;} \end{cases} \quad \Delta_P d = \begin{cases} 0_P & \text{if } d = \top_P, \\ d & \text{otherwise.} \end{cases}$$

Then the policy  $\nabla_P(\alpha, v)$  allows a request  $q$  whenever  $q$  contains a matching attribute, while the policy  $\Delta_P(\alpha, v)$  denies a request  $q$  whenever  $q$  contains a non-matching attribute value. Although the notion of conflicting policy decision has already been studied [5], to the best of our knowledge, this is the first time this notion of conflict has been used to evaluate targets. Intuitively, a conflict indicates that the request provides too much information for this particular policy.

**Proposition 16** *For all requests  $q$  and  $q'$  such that  $q \leq_{\mathcal{Q}_*} q'$  and for all atomic policies  $(\alpha, v)$ ,*

$$\text{Eval}_4(q, (\alpha, v)) \leq_4 \text{Eval}_4(q', (\alpha, v)).$$

A possible way to extend the operators defined in Table 1 is to consider the value  $\top_P$  as absorbing: for any operator  $\oplus : \mathbf{Three}^k \rightarrow \mathbf{Three}$ , we define the operator  $\hat{\oplus} : \mathbf{Four}^k \rightarrow \mathbf{Four}$  as follows:

$$\hat{\oplus}(d_1, \dots, d_k) = \begin{cases} \top_P & \text{if there exists } d_i = \top_P, \text{ for } 1 \leq i \leq k, \\ \oplus(d_1, \dots, d_k) & \text{otherwise.} \end{cases}$$

Clearly, given an operator  $\oplus$  defined over  $\mathbf{Three}$ , if  $\oplus$  is monotonic according to  $\leq_3$ , then  $\hat{\oplus}$  is also monotonic with respect to  $\leq_4$ . It follows that we can still safely use the operators generated by the operators  $\neg_P$ ,  $\wedge_P$ ,  $\vee_P$  and  $\bowtie_P$ , and we can deduce that any realizable policy is also monotonic. However, we lose the result of monotonic completeness, and we can no longer ensure that any monotonic operator can be generated from this set of operators. Obtaining such a result requires a deeper study of four-valued logic, and we leave it for future work.



## 5 Concluding Remarks

We have presented a generic framework for access control systems, within which a large variety of access control problems can be formalized, and which allows us to reason about the global properties of such systems. In particular, we have expressed the notions of monotonicity and completeness, and illustrated them using a particular instantiation of atomic policies that gives rise to a family of attribute-based access control models.

A major strength of our approach is that we do not provide “yet another access control language”. The framework is not intended to provide an off-the-shelf policy language or PDP (unlike XACML, for example), nor is it intended to be an access control model (in the style of RBAC96). Rather, we try to model all aspects of an access control system at an abstract level and to provide a framework that can be instantiated in many different ways, depending on the choices made for request attributes, atomic policies, policy decisions and policy evaluation functions. In doing so we are able to identify (i) how and why an access control system may fail to be sufficiently expressive, and (ii) how and why having an expressive access control system may lead to vulnerabilities. In the context of the framework, we are able to define what we mean by monotonic policies, which provide strong guarantees about policy evaluation when the requests may be missing information or contain erroneous information. We are also able to define what we mean by a complete system, which provides guarantees about the expressivity of the system.

Nevertheless our framework can be instantiated in ways that give rise to useful access control systems, and we have illustrated it with a monotonic and monotonically-complete attribute-based system, and with a complete attribute-based system. Moreover, the study of monotonicity and completeness in these concrete systems provides a better understanding of the issues that need to be considered when using those systems to specify authorization policies. In summary, the main technical contribution of this work is to provide a framework within which we can reason about important access control system properties such as expressivity and policy monotonicity. From a practical perspective, this work provides the developers of access control systems with a structured way of thinking about the design of such systems and to make them aware of the competing and conflicting considerations they need to consider.

There are many opportunities for future work, some of which we have discussed in previous sections. We believe that many other global properties of access control systems can be expressed within our framework and that tools from formal methods can be used to reason about the properties. For instance, in Section 3.3, we have outlined how the well-known *safety problem* in access control systems could be expressed within our framework. A very interesting question would be to identify the constraints on the components of our model for which problem becomes decidable. We also briefly sketch how we could perform static policy analysis, enabling us to check, for example, that a policy never returns  $\perp_P$  for well-formed requests. Again, this is a question of considerable interest in the community [5,11] and one that we believe could be addressed within our frame-

work. Our intuition suggests that this framework could also provide a basis for reasoning about privacy. It may be, for example, that a privacy-conscious user may wish or demand to withhold some attributes from an access control system. We conjecture that a property related to monotonicity will enable us to build useful access control systems that enable a user to disclose as little information as is required in order to gain access. Finally, in addition to instantiating this framework with other access control models, such as the different varieties of RBAC or XACML, we would like to study the composition of access control systems, and under what circumstances composition preserves properties such as monotonicity and completeness.

## References

1. D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model, Volume II. *Journal of Computer Security*, 4(2/3):229–263, 1996.
2. E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, 2003.
3. S. Blamey. *Handbook of Philosophical Logic*, volume 5, chapter Partial Logic, pages 261–353. Kluwer Academic Publishers, 2002.
4. P. Bonatti, S. De Capitani Di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security*, 5(1):1–35, 2002.
5. G. Bruns and M. Huth. Access control via Belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Transactions on Information and System Security*, 14(1):9, 2011.
6. J. Crampton and M. Huth. An authorization framework resilient to policy evaluation failures. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *ESORICS*, volume 6345 of *Lecture Notes in Computer Science*, pages 472–487. Springer, 2010.
7. J. Crampton and M. Huth. A framework for the modular specification and orchestration of authorization policies. In T. Aura, K. Järvinen, and K. Nyberg, editors, *NordSec*, volume 7127 of *Lecture Notes in Computer Science*. Springer, 2010.
8. J. Crampton and C. Morisset. PTaCL: A language for attribute-based access control in open systems. In P. Degano and J. D. Guttman, editors, *POST*, volume 7215 of *Lecture Notes in Computer Science*, pages 390–409. Springer, 2012.
9. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In M. Sloman, J. Lobo, and E. Lupu, editors, *POLICY*, volume 1995 of *Lecture Notes in Computer Science*, pages 18–38. Springer, 2001.
10. D. F. Ferraiolo and D. R. Kuhn. Role-based access control. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, 1992.
11. K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 196–205. ACM, 2005.
12. S. Jajodia, P. Samarati, M. Sapino, and V. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.
13. N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin. Access control policy combining: Theory meets practice. In *Proceedings of 14th ACM Symposium on Access Control Models and Technologies*, pages 135–144, 2009.

14. Q. Ni, E. Bertino, and J. Lobo. D-algebra for composing access control policy decisions. In *Proceedings of 2009 ACM Symposium on Information, Computer and Communications Security*, pages 298–309, 2009.
15. OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0*, 2005. OASIS Committee Specification (Tim Moses, editor).
16. OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0*, 2010. OASIS Committee Specification 01 (Erik Rissanen, editor).
17. D. Wijesekera and S. Jajodia. A propositional policy algebra for access control. *ACM Transactions on Information and System Security*, 6(2):286–235, 2003.
18. T. Y. C. Woo and S. S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

## A Proofs

**Proposition 11** *Let  $q, q' \in \mathcal{Q}$  such that  $q \equiv_{\mathcal{Q}_1} q'$ . Then for all atomic policies  $(\alpha, v)$ ,*

$$\text{Eval}(q, (\alpha, v)) = \text{Eval}(q', (\alpha, v)).$$

*Proof.* We can write  $q$  and  $q'$  as  $q_{\text{def}} \cup q_{\text{undef}}$  and  $q'_{\text{def}} \cup q'_{\text{undef}}$ , respectively, where  $q_{\text{undef}}$  is not necessarily equal to  $q'_{\text{undef}}$  but for all  $(\alpha, v) \in q_{\text{undef}}$  and all  $(\alpha', v') \in q'_{\text{undef}}$  we have  $v = v' = \circ$ . Then, since  $\text{Eval}(\{(\alpha, \circ)\}, (\alpha, v)) = \perp_P$  for all  $\alpha$  and all  $v$ , we have

$$\text{Eval}(q, (\alpha, v)) = \text{Eval}(q_{\text{def}}, (\alpha, v)) = \text{Eval}(q', (\alpha, v)).$$

**Proposition 12** *Let  $\text{Dec} = \{1_P, 0_P, \perp_P\}$ , and define  $x \leq_P y$  if and only if  $x = y$  or  $x = \perp_P$ . Then for all requests  $q$  and  $q'$  such that  $q \leq_{\mathcal{Q}} q'$  and for all atomic policies  $(\alpha, v)$ ,*

$$\text{Eval}(q, (\alpha, v)) \leq_P \text{Eval}(q', (\alpha, v)).$$

*Proof.* We reason by case over the value of  $\text{Eval}(q, (\alpha, v))$ .

- If  $\text{Eval}(q, (\alpha, v)) = \perp_P$ , the proposition clearly holds by definition of  $\leq_P$ .
- If  $\text{Eval}(q, (\alpha, v)) = 1_P$ , then by definition of  $\text{Eval}$ , we know that  $(\alpha, v) \in q$ , which implies by definition of  $\leq_{\mathcal{Q}}$  that  $(\alpha, v) \in q'$ , and thus we can conclude that  $\text{Eval}(q', (\alpha, v)) = 1_P$ .
- If  $\text{Eval}(q, (\alpha, v)) = 0_P$ , then by definition of  $\text{Eval}$ , there exists  $v' \in \text{dom}(\alpha)$  such that  $(\alpha, v') \in q$  and  $v' \neq v$ . By definition of  $\leq_{\mathcal{Q}}$ , it follows that  $(\alpha, v') \in q'$ , and since a request can only contain one value per attribute, we can conclude that  $\text{Eval}(q', (\alpha, v)) = 0_P$ .

**Proposition 16** *For all requests  $q$  and  $q'$  such that  $q \leq_{\mathcal{Q}} q'$  and for all atomic policies  $(\alpha, v)$ ,*

$$\text{Eval}(q, (\alpha, v)) \leq_P \text{Eval}(q', (\alpha, v)).$$

*Proof.* We reason by case over  $\text{Eval}(q, (\alpha, v))$ .

- If  $\text{Eval}(q, (\alpha, v)) = \perp_P$ , then we can trivially conclude.
- If  $\text{Eval}(q, (\alpha, v)) = 1_P$ , then by definition of  $\text{Eval}$ , we know that  $(\alpha, v) \in q$ , which implies by definition of  $\leq_{\mathcal{Q}}$  that  $(\alpha, v) \in q'$ . It follows that  $\text{Eval}(q', (\alpha, v))$  is equal either to  $1_P$  or to  $\top_P$ , and thus we can conclude.

- If  $\text{Eval}(q, (\alpha, v)) = 0_P$ , then by definition of  $\text{Eval}$ , there exists  $v' \in \text{dom}(\alpha)$  such that  $(\alpha, v') \in q$  and  $v' \neq v$ . By definition of  $\leq_Q$ , it follows that  $(\alpha, v') \in q'$ , and therefore either  $\text{Eval}(q', (\alpha, v)) = 0_P$  or  $\text{Eval}(q', (\alpha, v)) = \top_P$ , allowing us to conclude.
- If  $\text{Eval}(q, (\alpha, v)) = \top$ , then by definition of  $\text{Eval}$ ,  $(\alpha, v) \in q$  and there exists  $v' \in \text{dom}(\alpha)$  such that  $(\alpha, v') \in q$  and  $v' \neq v$ . By definition of  $\leq_Q$ , we have  $(\alpha, v) \in q'$  and  $(\alpha, v') \in q'$ , and thus we can conclude that  $\text{Eval}(q', (\alpha, v)) = \top$ .

## B Examples

We illustrate the framework with two well-known examples from the literature: an information-flow model [1] and a role-based access control (RBAC) model [10]. The examples are intended to illustrate the versatility of the framework. We assume the reader is familiar with the models concerned.

**Information Flow.** Consider an information flow policy [1] which states that a user request to read a file is authorized provided the security level associated with the user is at least as high as that associated with the file.

A user requests read access to a file then an authorization query could be presented to the PDP as a 4-tuple  $(u, f, b, \text{read})$ , where  $b$  is a flag that is set (by the context handler) if and only if  $\lambda(u) \geq \lambda(f)$ . We write  $\mathcal{Q}_{\text{IF}}$  for this set of queries. We also define  $\text{Dec}_{\text{IF}} = \{1_P, 0_P, \perp_P\}$ .

We define the set of atomic policies  $\mathcal{A}_{\text{IF}} = \{\text{allRead}, \text{isGtrThan}\}$ , such that **allRead** allows all read requests and **isGtrThan** permits any request in which the security level of the subject is greater than that of the object. The valuation for these policies is defined as:

$$\begin{aligned} \text{Eval}_{\text{IF}}(q, \text{allRead}) &= \begin{cases} 1_P & \text{if } q = (u, f, \text{read}, b), \\ \perp_P & \text{otherwise;} \end{cases} \\ \text{Eval}_{\text{IF}}(q, \text{isGtrThan}) &= \begin{cases} 1_P & \text{if } q = (u, f, a, 1), \\ \perp_P & \text{otherwise.} \end{cases} \end{aligned}$$

Note that  $\lambda(u) \not\geq \lambda(f)$  does not imply  $\lambda(u) < \lambda(f)$ , since the set of security levels may be a partial (not a total) order. Hence, it is not appropriate for  $\text{Eval}(q, \text{isGtrThan})$  to return  $0_P$  in the case that  $q$  is not of the form  $(u, f, a, 1)$ .

We introduce the set of operators  $\text{Ops}_{\text{IF}} = \{\wedge, ?, \sim\}$ , whose definitions can be found in Table 1. These operators allow us to define two different policies to address the information flow problem:  $p_1 = (\text{allowRead} \wedge \text{isGtrThan})$ , which (informally) says “if it is a read request and the security level of the user is greater than that of the file, then allow; otherwise deny”;  $p_2 = \text{allRead} ? (\sim \text{isGtrThan})$ , which returns the evaluation of  $\sim \text{isGtrThan}$  if it is a read request and  $\sim \text{isGtrThan}$

**Table 2.** Evaluation of information flow requests

	<b>allRead</b>	<b>isGtrThan</b>	$p_1$	$\sim \text{isGtrThan}$	$p_2$
$(u, f, \text{read}, 1)$	$1_P$	$1_P$	$1_P$	$1_P$	$1_P$
$(u, f, \text{read}, 0)$	$1_P$	$\perp_P$	$\perp_P$	$0_P$	$0_P$
$(u, f, \text{write}, 1)$	$\perp_P$	$1_P$	$\perp_P$	$1_P$	$\perp_P$

returns  $0_P$  whenever **isGtrThan** does not return  $1_P$ . We present in Table 2 some examples of evaluation of these policies.

Note that the difference between  $p_1$  and  $p_2$  is that  $p_2$  denies explicitly a request for read request for which the level of the user is not greater than that of the object, while  $p_1$  does not return a conclusive decision for such a request.

Using the previous definitions, we can define the system for information policies as:

$$S_{\text{IF}} = (\mathcal{Q}_{\text{IF}}, \{\text{allRead}, \text{isGtrThan}\}, \{1_P, 0_P, \perp_P\}, \{\wedge, ?, \sim\}, \text{Eval}_{\text{IF}})$$

Clearly, the policies  $p_1$  and  $p_2$  defined above are some of the policies realizable by  $S_{\text{IF}}$ , but many other policies can be generated from this system, for instance the policy  $(\sim \text{allRead})$ , that allows all read accesses and denies all other accesses.

This information-flow system can be extended to also write accesses by introducing two atomic policies **allWrite** and **isLessThan**, which allow write access and accesses where the level of the user is less than the one of the resource, respectively. Note that since  $\lambda(u) \not\geq \lambda(f)$  does not imply  $\lambda(u) < \lambda(f)$ , we would have to extend the request structure to also include a boolean attribute indicating if  $\lambda(u) \leq \lambda(f)$ . Due to space limitations, we do not describe the valuations of these atomic policies here, but they are clearly analogous to those for **allRead** and **isLessThan**. Finally, the complete information flow policy can be defined as  $((\text{allRead} ? (\sim \text{isGtrThan})) \Delta (\text{allWrite} ? (\sim \text{isLessThan})))$ .

**RBAC.** The role-based access control (RBAC) model [10] introduces the notion of roles, to which users and permissions are assigned. A request, modeled as a session-permission pair, is allowed if and only if the session – a subset of the roles assigned to the user associated with the session – contains at least one role that is assigned to the permission.

In our framework, a request has the form  $(\{r_1, \dots, r_k\}, prm)$ , where  $s = \{r_1, \dots, r_k\}$  is a session and  $prm$  is a permission. We write  $\mathcal{Q}_{\text{RBAC}}$  for the set of such requests. We define  $\text{Dec}_{\text{RBAC}} = \{1_P, 0_P\}$ .

We define one atomic policy for each pair  $(r, prm) \in PA$ , where  $PA$  is the permission-role assignment relation. The valuation function for an atomic policy is therefore quite straightforward:

$$\text{Eval}_{\text{RBAC}}((\{r_1, \dots, r_k\}, prm), (r, prm')) = \begin{cases} 0_P & \text{if } r \notin s \text{ or } prm \neq prm', \\ 1_P & \text{otherwise.} \end{cases}$$

Given a relation  $PA$ , the corresponding RBAC policy is simply the disjunction of all atomic policies. Hence, it is sufficient to limit the set of operators to  $\text{Dec}_{\text{RBAC}} = \{\vee\}$ . Note that it means it is also possible to generate a less permissive policy, by excluding some atomic policies. The access control system for RBAC is given by:

$$S_{\text{RBAC}} = (\mathcal{Q}_{\text{RBAC}}, PA, \{1_P, 0_P\}, \{\vee\}, \text{Eval}_{\text{RBAC}})$$